NSWI170 – **Počítačové systémy** – 2023/24 LS – Martin Svoboda

Úkol **A2: Kulička**

Závazné pokyny | Dobře míněné rady | Náměty k zamyšlení | Časté chyby

201. [**Dekompozice do funkcí a tříd**] Veškerý kód dekomponujeme do vhodně navržených globálních funkcí, tříd a jejich členských funkcí (metod), a to na různých úrovních abstrakce od uživatelsky přívětivého ovládání jednotlivých diod až po řešení logiky samotné běhající kuličky.

202. [**Třída pro reprezentaci diody**] Jednotlivé diody budeme reprezentovat jako instance třídy, která zapouzdří všechny potřebné datové položky i očekávanou funkcionalitu. Cílem tedy není nic jiného než navrhnout ovladač diody, který umožní jeho uživatelsky přívětivé používání, nabídne ucelené rozhraní a současně záměrně skryje vnitřní technické a implementační detaily.

203. [**Privátní datové položky**] Dobrou praxí při návrhu tříd našeho druhu je, že všechny jejich datové položky učiníme privátními. Manipulace s nimi tedy nebude možná přímo zvenku, ale jen nepřímo prostřednictvím veřejných metod, které za tímto účelem navrhneme. Díky tomu budeme mít jejich obsah pod kontrolou, jinak by je totiž mohl měnit kdokoli a odkudkoli bez našeho vědomí. Obdobně bychom za privátní měli označit i případné metody, které jsou čistě jen vnitřního charakteru.

204. [**Veřejné rozhraní diody**] Třída diody nabídne minimálně dvě veřejné funkce, jednu na inicializaci dané diody a druhou na změnu jejího stavu rozsvícení. Jejich úkolem je zabalit příslušná nízkoúrovňová systémová volání Arduina, čímž uživatelům umožníme ovládání diod na vyšší úrovni abstrakce.

205. [**Logická čísla diod**] Abychom opravdu vyšší abstrakce mohli dosáhnout, budeme všude v kódu používat výhradně logická čísla diod `0` až `3` namísto nízkoúrovňových pinů. Tato čísla pak v tomto pořadí budou odpovídat diodám s označeními `D1` až `D4`, a tedy pinům `led1_pin` až `led4_pin`. Jedinou výjimkou, kde můžeme pracovat přímo s piny, budou právě dvě výše zmíněné funkce, které jako jediné budou interně provádět potřebný překlad logických čísel na odpovídající piny.

206. [**Hodnoty čísel pinů**] Nahlédnutím do přiloženého hlavičkového souboru se sice můžeme snadno dopátrat, že čísla pinů jednotlivých diod mají hodnoty `13` až `10`, určitě s těmito konkrétními čísly ale nemůžeme jakkoli přímo nebo i nepřímo pracovat. Nemůžeme ani předpokládat, že jsou případně nějak uspořádaná nebo že tvoří souvislý interval.

207. [**Nezávislost na hodnotách konstant**] Tuto myšlenku můžeme zobecnit a rozšířit i na práci s jakýmikoli jinými konstantami, které nemáme plně pod kontrolou. Pokud nám jejich autor explicitně nepřislíbil nějaké speciální záruky, nemůžeme jednoduše vůči jejich konkrétním hodnotám uplatňovat žádné předpoklady. Musíme tedy na nich být zcela nezávislí.

208. [**Variabilní počet diod**] Celá aplikace musí být naprogramována tak, abychom nikde nevynucovali ani nepředpokládali nějaký konkrétní fixní počet diod. Jinými slovy vše musí být univerzální a fungovat i při jiném počtu diod než zrovna `4`. Konkrétní počet dostupných diod pak odvodíme z velikosti námi vytvořeného pole, které budeme používat na překlad logických čísel diod na jejich piny.

209. [**Počáteční vypnutí diod**] V rámci inicializace Arduina je nutné kromě nastavení módů pinů zařídit i vypnutí všech diod, protože obecně nemáme zaručeno, v jakém aktuálním stavu se mohou vyskytovat.

210. [**Předčasná systémová volání**] Přestože bychom se do takové situace s našimi prostředky vlastně ani neměli dostat, zdůrazněme, že nesmíme žádné systémové funkce Arduina volat před funkcí `setup`, tedy před jeho inicializací. Zatímco na reálném Arduinu by taková volání obvykle prošla bez jakéhokoli efektu, emulátor v ReCodExu takové situace hlídá striktně.

211. [**Konstruktory objektů diod**] Výše popsaná komplikace by specificky mohla nastat v situaci, kdy bychom pro naše objekty diod chtěli definovat explicitní konstruktory, v rámci nichž bychom např.

chtěli nastavovat módy pinů. Instance objektů globálních proměnných se totiž vytváří už na začátku běhu programu, tedy před funkcí `setup`. Pokud se tudíž takové konstruktory rozhodneme používat, musíme si být tohoto chování vědomi.

212. [**Implicitní konstruktory**] Mnohem lepší ale bude, pokud explicitní konstruktory vůbec používat nebudeme. Přineslo by nám to totiž jen další technické komplikace bez jakýchkoli výhod a ani jsme se je stejně neučili. Jinými slovy se spolehneme na implicitní konstruktory, které za nás vytvoří kompilátor sám. Ty zařídí vytvoření příslušného objektu v takové podobě, že jeho datové položky nebudou nijak inicializovány. Nastavení jejich počátečních hodnot si tedy zařídíme sami přes nějakou naši inicializační metodu volanou z funkce `setup`.

213. [**Vytváření instancí tříd**] Pokud chceme pomocí takového bezparametrického konstruktoru vytvořit nový objekt diody, stačí jen napsat `Diode diode`. Klíčové slovo `new` nepíšeme, to má v jazyce C++ úplně jiný význam. Dodejme také, že proměnná `diode` skutečně obsahuje instanci diody jako takové, nikoli ukazatel nebo nějakou referenci na ní.

214. [**Ovládání diod**] Třída pro diodu nabídne minimálně metody, jejichž prostřednictvím dokážeme danou diodu zapnout nebo vypnout. Není ale vhodné, abychom k tomuto účelu nabídli dvě samostatné metody ve stylu `on` a `off`, protože by pak volající nedokázal dynamicky vyjádřit svůj požadavek, aniž by musel svůj kód nějak podmíněně větvit. Lepší tedy bude nabídnout metodu s `true` / `false` parametrem, který rozlišení záměru zapnutí / vypnutí snadno umožní.

215. [**Zachování stavu diody**] Dodejme také, že dokud dané diodě nedáme nový nízkoúrovňový pokyn k rozsvícení / zhasnutí přes volání funkce `digitalWrite`, její stav zůstane zachovaný beze změny.

216. [**Reprezentace vnitřního stavu**] Pokud se rozhodneme pamatovat si nebo jinak vyjadřovat aktuální stav diody, je potřeba jej reprezentovat na logické úrovni. Ideálně tedy např. pomocí hodnot typu `bool`, nikoli pomocí nízkoúrovňových konstant `LOW` a `HIGH`. Takové logické stavy pak také na uvedené konstanty typu `int` musíme umět korektně transformovat.

217. [**Pojmenované konstanty**] Všechny konstanty mající nějaký logický význam ve vztahu k našemu programu je nutné deklarovat a vhodně pojmenovat. Je-li to navíc možné s ohledem na jiné již definované konstanty, je současně nutné jejich hodnotu pomocí takových konstant i spočítat či jinak odvodit, a to samozřejmě korektně.

218. [**Příznak konstantnosti**] Také nezapomeňme na to, abychom samotný příznak konstantnosti jako takové uvedli u každé datové položky, která takový charakter má.

219. [**Globální proměnné**] S ohledem na programovací model nabízený Arduinem je nutné, abychom si řadu informací pamatovali prostřednictvím globálních proměnných, přestože takové řešení by v případě standardních aplikací vhodné nebylo. Jinak bychom je totiž nemohli mezi funkcí `setup` a jednotlivými voláními funkce `loop` přenášet. Počet takových proměnných však budeme držet na nezbytném minimu, zejména nebudeme používat globální proměnné v situacích, kde by ve skutečnosti stačily jen obyčejné lokální.

220. [**Přístup ke globálním proměnným**] Globální proměnné budeme považovat za natolik významné, že k nim budeme přistupovat napřímo, a nebudeme je tedy předávat přes parametry funkcí. Ono by to totiž v případě složitějších objektů jako našich diod ani s našimi znalostmi nešlo, protože by docházelo k vytváření jejich kopií, které by se od původních objektů zcela oddělily.

221. [**Jednoduchý obsah loopu**] Ať už logiku naší pohybující se kuličky dekomponujeme jakkoli, je nutné zajistit, že kód uvedený přímo v těle funkce `loop` nebude příliš rozsáhlý, komplikovaný nebo technický. V ideálním případě bychom zde měli řešit jen nějaké základní operace na co nejvyšší úrovni abstrakce, podobně jako tomu bylo u funkce `main`.

222. [**Efektivní implementace loopu**] Funkce `loop` tvoří jádro našeho programovacího modelu, má tedy klíčový dopad na efektivitu celé naší aplikace. A nejde samozřejmě jen o funkci samotnou, ale i všechny další funkce, které z ní budeme přímo či nepřímo volat. Samotná funkce `loop` je totiž vykonávána neustále pořád dokola, a to přibližně 1000× do sekundy. V závislosti na složitosti programu to však může být i o řád častěji nebo naopak méně často.

223. [**Třída pro reprezentaci časovače**] Funkcionalitu pro časování událostí také vyřešíme pomocí vhodně navržené samostatné třídy. Bude se nám to totiž hodit v budoucnu, kdy budeme potřebovat využívat dokonce více paralelně běžících a navzájem nezávislých časovačů najednou.

224. [**Funkcionalita časovače**] Oceníme také možnost postupně používat měnící se hodnoty očekávaných intervalů, stejně jako časovač inicializovat (spustit) kdykoli dle potřeby (nejenom ve funkci `setup`). Konečně dodejme, že časovač by měl být jen pasivní prostředek nabízející nám kontrolu časování skrze elegantní rozhraní, sám ale žádné akce vyvolávat nemá.

225. [**Mechanismy kontroly časování**] V obecné rovině se nám pro kontrolu časování nabízí v zásadě dva mechanismy. Buď si zapamatujeme čas předchozí události a budeme detekovat, jestli už od ní uplynul požadovaný interval, nebo si rovnou naplánujeme čas příští události a budeme detekovat, jestli už nastal. Jakkoli každý přístup nabízí určité výhody i nevýhody, první varianta pro nás bude vhodnější, flexibilnější i technicky jednodušší.

226. [**Detekce uplynutí času**] Samotnou kontrolu uplynutí požadovaného časového intervalu je potřeba vyřešit obezřetně. Jak už jsme diskutovali, jedna iterace funkce `loop` sice může trvat podstatně kratší dobu než 1 ms, stejně tak ale i delší. Tím pádem nemáme garantováno, že budeme schopni zamýšlený časový okamžik další události detektovat s absolutní přesností, tedy na úrovni jednotlivých milisekund.

227. [**Čas předchozí události**] Pokud nejsme schopni konkrétní časové okamžiky detekovat přesně, bylo by chybou uložit si skutečný aktuální čas právě vyvolané události za účelem zapamatování si času předchozí události. Kdybychom to totiž udělali, systematicky by mohlo docházet k opakovanému zpožďování, které by se nám postupně akumulovalo a narušilo by celkovou pravidelnost událostí.

228. [**Zjišťování aktuálního času**] Volání funkce `millis` sice není časově přespříliš náročné, stejně tak ale ani rychlé. Z tohoto důvodu je vhodné zajistit, že ji v rámci jedné iterace funkce `loop` zavoláme pouze a jen jednou a takto získanou hodnotu pak prohlásíme za aktuální čas platný pro celou danou iteraci. To nám navíc přinese i další výhody. Při opakovaném zjišťování aktuálního času bychom totiž mohli dostat jiné hodnoty. To by pak v závislosti na našem kódu mohlo přinést i velmi nepříjemné chybové situace způsobené takovou nekonzistencí.

229. [**Přetečení hodnoty času**] Pro reprezentaci systémového času se používá datový typ `unsigned long`. Ten nám umožní uchovat čas v milisekundách odpovídající až necelým 50 dnům, i tak ale dříve nebo později k přetečení jeho hodnoty nevyhnutelně dojde. I takovou situaci tudíž musíme umět korektně ošetřit.

230. [**Velikosti datových typů**] Jen tak mimochodem, kdykoli tedy chceme pracovat s hodnotami času, měli bychom důsledně používat zmíněný typ `unsigned long`. Už i u `signed` varianty a tím spíše u typu `int` bychom totiž riskovali problémy s menším rozsahem. Dodejme také, že velikosti těchto typů se mohou lišit podle platformy, takže zatímco `long` má obvykle 8 B, na Arduinu je to jenom 4 B.

231. [**Aritmetika nad čísly diod**] S proměnnými bychom vždy měli pracovat v souladu s jejich datovým typem, ale také očekáváními vyplývajícími z jejich logické podstaty. Pokud například pozice diod reprezentujeme pomocí jejich logických čísel, nemůžeme nad nimi řešit násobení nebo počítat absolutní hodnotu, jakkoli z technického pohledu na věc takové operace celá čísla jako taková připouští.

232. [**Zákaz neplatných hodnot**] Analogicky také nemůžeme pracovat s pozicemi jako `-1`, které ani samy o sobě nejsou validní. Tuto myšlenku pak můžeme zobecnit tak, že ani v jiných situacích nikdy nebudeme vyčleňovat a používat nějaké jinak normální hodnoty pro reprezentaci chybných, okrajových nebo třeba neočekávaných situací, hodnot apod.

233. [**Realizace prvotní akce**] V rámci funkce `setup` se očekávají nejrůznější inicializační akce, včetně např. nastavení počáteční pozice naší kuličky. Ať už samotné rozsvícení odpovídající diody provedeme hned, nebo jej jednoduchým trikem odložíme a provedeme až v rámci standardní běhové logiky naší aplikace ve funkci `loop`, první rozsvícenou diodou by měla být dioda číslo `0`.

234. [**Inicializace časovače**] Pokud jde o nastavení výchozí hodnoty časovače, uvědomme si, že aktuální čas v průběhu funkce `setup` už nutně nemusí být roven `0`.

235. [**Zombie akce v loopu**] V návaznosti na požadavek efektivity funkce `loop` je potřeba se vždy velmi pečlivě zamyslet nad tím, jaké akce v rámci ní chceme provádět. Nepřijatelné by například bylo, pokud bychom do ní vložili nějakou podmíněnou akci, u které bychom ale ve skutečnosti už předem věděli, že bude provedena jen jednou jedinkrát hned v první iteraci a pak už nikdy.

236. [**Význam aktuální pozice**] Pro zvládnutí logiky pohybující se kuličky budeme patrně muset nějak řešit její aktuální polohu. Měli bychom se tudíž zamyslet nad tím, co přesně má v tomto kontextu slovo *aktuální* vlastně znamenat. Jinými slovy bychom se měli ujistit, že onou aktuální polohou ve skutečnosti nemyslíme například polohu předcházející nebo naopak následující.

237. [**Šablona pozic kuličky**] Přestože by vlastně šlo o hezký nápad, náš problém pohybu kuličky nebudeme řešit tak, že bychom si vytvořili jakousi šablonu posloupnosti očekávaných poloh naší kuličky, přes které bychom následně iterovali. Neodpovídá to totiž logice našeho zadání, které musíme pochopitelně vždy respektovat.

238. [**Logické pořadí akcí**] Při realizaci vlastního pohybu kuličky je více než vhodné respektovat intuitivní pořadí jednotlivých akcí, tedy nejprve vypnout stávající diodu, následně provést přesun na další pozici a teprve pak novou diodu zapnout. Odlišná uspořádání by byla zavádějící.

239. [**Nepovolené systémové funkce**] Celá naše aplikace musí být naprogramována takovým způsobem, abychom se obešli bez použití funkcí `delay` nebo `delayMicroseconds`, stejně jako nesmíme ani žádným jiným způsobem blokovat průběh funkce `loop`. Jednoduše proto, že by takovéto snahy šly přímo proti logice celého našeho programovacího modelu.

240. [**Chybové signály**] Pokud během ladění programu v ReCodExu narazíme na chybový signál (což je něco jiného než návratový kód), došlo při běhu programu k natolik závažnému problému, že musel být ukončen. Příčiny mohou být následující: signál `4` (dělení nulou nebo jiná neplatná instrukce), `6` (systémová volání před funkcí `setup`), `9` (nedostatek paměti) nebo `11` (přístup do nealokované paměti, např. na položky mimo pole).

241. [**Návaznost úkolů**] Zbývající úkoly na sebe budou bezprostředně navazovat a komponenty jako naše diody, časovač nebo v budoucnu např. tlačítka budeme potřebovat využívat opakovaně. Je proto vhodné na ně pohlížet jako na menší součásti nějakého složitějšího postupně budovaného projektu. Už od samotného začátku se vyplatí nepodcenit jejich návrh a naopak vše implementovat v co nejvyšší kvalitě, abychom se k nim už později nemuseli vracet. Jakékoli další zásahy v podobě rozšíření nebo dokonce oprav totiž budou časově mnohem náročnější a přinesou jen riziko zanesení nových a obtížně odladitelných problémů.